设计文档

> 不要因为别人交卷就乱答题
>
> 详细记录！设计好再开写。高老板的忠告

1.数据通路

本流水线CPU为五级流水线结构。功能部件规划如下：F(IFU), D(NPC, RF, EXT, CMP), E(ALU), M(DM), W(RF)。

1.1.分布式译码

由F_cu，D_cu，E_cu，M_cu，W_cu对当前流水段进行译码分析。以下是CU的说明：

CU

端口说明

信号名称	方向	描述
opcode[5:0]	I	Instr[31:26]
func[5:0]	I	Instr[5:0]
NPCOp[3:0]	O	NPC的控制信号
CMPOp[2:0]	O	CMP的控制信号
RFWrEn	O	RF的写使能
RFWRSel[2:0]	O	RF写寄存器的选择信号
RFWDSel[2:0]	O	RF写数据的控制信号
EXTOp[1:0]	O	EXT的控制信号
Start	O	指令是否为四种乘除信号之一
MDUOp[3:0]	O	MDU的控制信号
ALUOp[3:0]	O	ALU的控制信号
ALUASel[1:0]	O	ALUA的选择信号
ALUBSel[1:0]	O	ALUB的选择信号
DMWrEn	O	DM的写使能
DMOp[1:0]	O	DM的控制信号
DMEXTOp	O	DM的符号扩展控制信号
calc_r	O	指令为“ALU计算&R”型
calc_i	O	指令为“ALU计算&I”型
lui	O	指令为“Lui”型
md	O	指令为“乘除法”型
mt	O	指令为“写HILO型”
mf	O	指令为“读HILO且写寄存器”型
load	O	指令为“Load”型
store	O	指令为“Store”型
branch	O	指令为“Branch”型
j_l	O	指令为“跳转-链接”型
jr	O	指令为“跳转至寄存器型”

真值表

lui的结果产生移至D级的EXT，E_E32即结果。利用ALU的ADD和GRF[0]相加，在M级和M_AO合流。

	NPCOp[3:0]	RFWrEn	RFWRSel[2:0]	RFWDSel[2:0]	EXTOp	ALUOp[3:0]	CMPOp[3:0]	ALUASel[1:0]	ALUBSel[1:0]	DMWrEn	DMOp[1:0]	DMEXTOp
add	NPCOp_PC4	1	RFWRSel_rd	RFWDSel_ALU		ADD	CMPOp_non	ALUASel_rs	ALUBSel_rt			
sub	NPCOp_PC4	1	RFWRSel_rd	RFWDSel_ALU		SUB	CMPOp_non	ALUASel_rs	ALUBSel_rt			
ori	NPCOp_PC4	1	RFWRSel_rt	RFWDSel_ALU	EXTOp_zero	OR	CMPOp_non	ALUASel_rs	ALUBSel_imm			
lui	NPCOp_PC4	1	RFWRSel_rt	RFWDSel_ALU	EXTOp_lui	ADD	CMPOp_non	ALUASel_rs	ALUBSel_imm			
lw	NPCOp_PC4	1	RFWRSel_rt	RFWDSel_DMRD	EXTOp_sign	ADD	CMPOp_non	ALUASel_rs	ALUBSel_imm		DMOp_w	
sw	NPCOp_PC4	0			EXTOp_sign	ADD	CMPOp_non	ALUASel_rs	ALUBSel_imm	1	DMOp_w	
jal	NPCOp_JL	1	RFWRSel_31	RFWDSel_PC8			CMPOp_non	ALUASel_rs				
jr	NPCOp_JR	0					CMPOp_non	ALUASel_rs				
beq	NPCOp_Br	0					CMPOp_beq	ALUASel_rs	ALUBSel_rt			

1.2.功能部件

1.2.1.Stall

Stall内部三次实例化CU，对三级指令译码，解出T_{use}和T_{new}。

1.2.1.1.阻塞条件

- $T_{new}>T_{use}$
- 用的寄存器和写的寄存器是同一个，即D_rs_addr==A3
- 用的寄存器不为$0

端口说明

信号名称	方向	描述
D_Instr	I	D级指令
E_Instr	I	E级指令
M_Instr	I	M级指令
E_A3	I	E级指令写的寄存器
M_A3	I	M级指令写的寄存器
Start	I	乘除法指令开始标志
Busy	I	乘除法正在进行的标志
Stall	O	高电平暂停

1.2.1.2.暂停实现

```
Stall = Stall_rs || Stall_rt || Stall_md
```

- Stall_md：当**Busy或Start为1**时，若D级为读写HILO的信号，即**D_Mt或D_Mf为1**时，**阻塞。**
- Stall_rs或Stall_rt：使用AT法，如下：

在D级暂停，加入气泡。因此要给ifu和FD_REG, DE_REG加上使能信号。DE_REG中：当使能信号为低电平时，将E_Instr赋值为nop。

需求者的最晚时间模型·T_{use}

T_{use}：指令进入D级后，其后的某个功能部件再经过多少cycle就**必须**要使用寄存器值

供给者的最早时间模型·T_{new}

T_{new}：位于E级及其后各级的指令，再经过多少cycle能够产生要写入寄存器的结果。

暂停转发的基本方法

A指令位于D级，将A的T_{use}与位于E/M/W各级指令的T_{new}比较。若T_{new}大，则暂停，否则转发。

暂停机制构造方法

注意事项

- 只关注每条指令的操作语义
- 指令可能有2个不同的T_{use}

真值表

- T_{use}：无定义处应取极限大值。避免stall误触发。

指令	T_{use}^{rs}	T_{use}^{rt}
calc_r	1	1
calc_i	1	3
lui	0	0
md	1	1
mt	1	3
mf	3	3
load	1	3
store	1	2
branch	0	0
jr	0	3

- T_{new}：无定义处应取0。避免stall误触发。

指令类型	T_{new}@E	T_{new}@M	T_{new}@W
calc_r	1	0	0
calc_i	1	0	0
lui	0	0	0
md	0	0	0
mt	0	0	0
mf	1	0	0
load	2	1	0
store	0	0	0
branch	0	0	0
j_l	0	0	0

1.2.2.F_PC

端口说明

信号名称	方向	描述
WE	I	写使能(!Stall)
clk	I	时钟信号
reset	I	同步复位
NPC	I	来自NPC的次地址
PC	I	更新后的地址

1.2.3.FD_REG

端口说明

信号名称	方向	描述
F_Instr	I	F级指令
F_PC	I	F级地址
D_Instr	O	D级指令
D_PC	O	D级地址
clk	I	时钟信号
reset	I	同步复位
WE	I	写使能(!Stall)
flush	I	高电平时清空延迟槽。flush=CMPOp==`CMPOp_Bxxzall && !D_b_jump

1.2.4.D_CU

控制NPC, RF, EXT, CMP。

需要译码得到RFWRSel，以此确定A3。当得到了写寄存器的编号，就可以判断转发。所以A3参与流水。

WD不参与流水，因为它具有不确定性，故让其MUX数据集进入流水。

端口说明

信号名称	方向	描述
opcode	I	D_Instr[31:26]
func	I	D_Instr[5:0]
NPCOp	O	NPC的控制信号
RFWRSel	O	RF-A3的选择信号
EXTOp	O	EXT的控制信号
CMPOp	O	CMP的控制信号

1.2.5.D_NPC

B指令和**J指令**支持延迟槽，所以使用PC@F+4。此次设计不采用直接输出PC8的格式，而是在PC的流水中，随用随加8。

同时接受F_PC和D_PC的输入。当D指令的NPCOp为顺序default值时，NPC=**F**_PC+4；当D指令的NPCOp为B类时，NPC采用PC@**F**+4；当D指令为JL类，NPC只与**D**相关；最后当D指令为JR类时，NPC=RS。

输入RS被转发处理过。

NPC输出**直接回写F级。**

端口说明

信号名称	方向	描述
F_PC	I	F级指令
NPCOp	I	D级指令对应的NPC控制信号
IMM26	I	26位立即数
IMM16	I	16位立即数
RS	I	jr需要的寄存器内容
Branch	I	B指令跳转选择
NPC	O	回写F级IFU

控制信号

NPCOp	描述
NPC_PC4	顺序
NPC_Br	分支
NPC_JL	跳转并链接
NPC_JR	跳转寄存器内容

1.2.6.D_RF

只实例化一次，D级只使用读功能。写功能的部署在W级。

回写：操作在RF中进行，输出在`mips.txt`中进行。

- **内部转发**：当读和写同一个寄存器时，**读出的数据应该为要写入的数据。**
- 输出D_V1和D_V2**经过转发处理，或许进入流水。**

端口说明

信号名称	方向	描述
clk	I	时钟信号
reset	I	同步复位
A1	I	rs寄存器编号，D_Instr[25:21]
A2	I	rt寄存器编号，D_Instr[20:16]@D
RD1	O	D_V1
RD2	O	D_V2
W_Instr	I	W级指令
RFWrEn	I	W级指令控制
A3	I	W_A3
WD	I	W_RW

1.2.7.D_EXT

输出D_E32进入流水。

端口说明

信号名称	方向	描述
IMM16	I	16位立即数
EXTOp	I	EXT的控制信号
IMMEXT	O	32位扩展立即数D_E32 / lui结果

控制信号

EXTOp	描述
EXTOp_zero	0扩展
EXTOp_sign	符号扩展
EXTOp_lui	lui计算

1.2.8.D_CMP

端口说明

信号名称	方向	描述
D1	I	经过转发处理的GRF[rs]，即FD_rs
D2	I	经过转发处理的GRF[rt]，即FD_rt
CMPOp	I	CMP的控制信号
Branch	O	NPC的跳转信号

控制信号

CMPOp	描述
CMPOp_bep	beq比较
CMPOp_bne	bne比较

1.2.9.DE_REG

输出E_A3进入流水，E_V2经过转发处理，或许进入流水。

控制信号

信号名称	方向	描述
WE	I	写使能，默认为1
flush		阻塞时插入nop->flush=Stall
D_b_jump	I	D级B指令是否跳转
E_b_jump	O	
D_V1	I	经过转发处理的GRF[rs]，即FD_rs
D_V2	I	经过转发处理的GRF[rt]，即FD_rt
D_E32	I	32位扩展立即数 / **lui的计算结果**
D_Instr	I	指令
D_PC	I	地址
D_A3		写寄存器编号
E_Instr	O	
E_PC	O	
E_V1	O	GRF[rs]
E_V2	O	GRF[rt]
E_E32		32位扩展立即数 / **lui的计算结果**
E_A3		
clk	I	时钟信号
reset	I	同步复位

1.2.10.E_CU

端口说明

信号名称	方向	描述
opcode	I	E_Instr[31:26]
func	I	E_Instr[5:0]
Start	O	乘除法标志
MDUOp	O	MDU控制信号
ALUOp	O	ALU控制信号
ALUASel	O	操作数A选择信号
ALUBSel	O	操作数B选择信号

1.2.11.E_ALU

输出AO进入流水。

端口说明

信号名称	方向	描述
A	I	经过转发处理和MUX选择的操作数A
B	I	经过转发处理和MUX选择的操作数B
ALUOp	I	ALU的控制信号
ALU_RES	O	ALU的结果

控制信号

ALUOp	描述
ADD	加法
SUB	减法
OR	或运算
AND	与运算
SLT	有符号比较置一
SLTU	无符号比较置一

1.2.12.E_MDU

HI和LO、ALU_RES用三目运算符，合流进E_AO。同时改ALU输出为E_ALU_RES。

端口说明

信号名称	方向	描述
A	I	GRF[rs]
B	I	GRF[rt]
LO	O	乘法：低32位；除法：商
HI	O	乘法：高32位；除法：余数
Start	I	指令是否为四种乘除之一
Busy	O	高电平：正在进行乘除法
clk	I	时钟信号
reset	I	同步复位
MDUOp	I	MDU的控制信号

1.2.13.EM_REG

端口说明

信号名称	方向	描述
E_b_jump	I	E级B指令是否跳转
M_b_jump	O	
E_V2	I	经过转发处理的GRF[rt]，即FE_rt
E_AO	I	ALU的结果
E_Instr	I	指令
E_PC	I	地址
E_A3	I	写寄存器编号
M_V2	O	
M_AO	O	
M_Instr	O	
M_PC	O	
M_A3	O	
clk	I	时钟信号
reset	I	同步复位

1.2.14.M_BE

SJudge和LJudge的工作：

- SJudge：根据DMOp，决定DMByteEn高电平位和WD有效位。注意：DMByteEn的第`i`位为高电平=m_data_rdata的第`i`个字节写入fixed_data的第`i`个字节。这和`sb`、`sh`的定义不太一样，所以要将生数据处理，将有效位移到有效字节范围。
- LJudge：根据DMOp和DMEXTOp，将DM读出的生数据进行处理，后输出。

端口说明

信号名称	方向	描述
WE	I	写使能（Store指令触发，作用域：SJudge）
WD_raw	I	未经处理的DM写数据(FM_DW)
WD	O	经过处理的DM写数据（m_data_rdata）
DMRD_raw	I	DM读出的32位生数据
DMAddr	I	DM写地址
DMOp	I	字节操作的控制信号
DMEXTOp	I	字节符号扩展的控制信号
DMByteEn	O	字节使能
DMRD	O	经过字节操作和符号扩展的数据

1.2.15.MW_REG

端口说明

信号名称	方向	描述
M_b_jump	I	M级B指令是否跳转
W_b_jump	O	
M_AO	I	ALU计算出的结果
M_DR	I	DM读出的结果
M_Instr	I	指令
M_PC	I	地址
M_A3	I	写寄存器编号
W_AO	O	
W_DR	O	
W_Instr	O	
W_PC	O	
W_A3		
clk	I	时钟信号
reset	I	同步复位

1.2.16.W_CU

端口说明

信号名称	方向	描述
opcode	I	W_Instr[31:26]
func	I	W_Instr[5:0]
RFWrEn	O	RF的写使能
RFWDSel	O	RF的写数据选择

1.2.17.Forward

外部转发：mips转发区（FD_rs，FD_rt，FE_rs，FE_rt，FM_DW）

内部转发：RF寄存器内部。当读写寄存器相同且编号不为0时进行内部转发。

需求者

- RF内部读写编号相同时的输出端口（内部转发：RFWD->RFRD）
- CMP的两个输入端口
- ALU的两个输入端口
- NPC的RA输入端
- DM的写数据输入端
- DE_REG, EM_REG传递的寄存器值

选择数据

借用mips.v中的cu进行译码，确定供给者是否转发（信号：Forward）。

采用**就近原则**选择级次转发的数据。

转发输出(优先级：高->低)	恰在本级产生寄存器结果的指令	供给者
DE_REG	J_L / LUI	E_PC+32'd8 / E_E32
EM_REG	CALC_R / CALC_I / D_Mf	M_PC+32'd8 / M_AO
MW_REG	LOAD	W_RW
寄存器内部转发		

接受条件

- 供给端的寄存器地址与当前的相同
- 当前需要的地址不为0
- 供给端可以转发 Forward

2.注意

位宽一致

DMOp的位宽没匹配上，导致bug

3.测试方案

数据生成

构造以四为单位的连续冒险阵列。考虑到跳转很难处理。所以先生成除跳转以外的测试序列，然后手动将跳转安插进去。

```
import random
import sys
calc_r_size = 5
calc_i_size = 2
md_size = 3
mft_size = 3
calc_r = ["add","sub","and","or","slt","sltu"]
calc_i = ["addi","andi","ori"]
md = ["mult","multu","div","divu"]
mft = ["mfhi","mflo","mthi","mtlo"]

# 0.open test.asm
record_path = "C:\\Users\\shael\\CO\\P6\\test\\res\\record"
sys.stdout = open(record_path+"\\test.asm", "w")

# 1.instruction declare
def inst_calc_r(rd,rs,rt):
    print(calc_r[random.randint(0,calc_r_size)]+"
$"+str(rd)+",$"+str(rs)+",$"+str(rt))
    return
def inst_calc_i(rt,rs,imm):
    print(calc_i[random.randint(0,calc_i_size)]+"
$"+str(rt)+",$"+str(rs)+","+str(imm))
    return
def lui(rt,imm):
    print("lui $"+str(rt)+" "+str(imm))
def inst_md(rs,rt):
    mdop = random.randint(0,0)
    match mdop:
        case 0:
            print(md[random.randint(0,1)]+" $"+str(rs)+",$"+str(rt))
        case 1:
            print(md[random.randint(2,3)]+" $"+str(rs)+",$"+str(rt))
    return
def inst_mft(rs):
    print(mft[random.randint(0,mft_size)]+" $"+str(rs))
    return
def lb(rt,offset,base):
    print("lb $"+str(rt)+", "+str(offset)+"($"+str(base)+")")
    return
def lh(rt,offset,base):
    print("lh $"+str(rt)+", "+str(offset)+"($"+str(base)+")")
    return
def lw(rt,offset,base):
    print("lw $"+str(rt)+", "+str(offset)+"($"+str(base)+")")
    return
```

```
def sb(rt,offset,base):
    print("sb $"+str(rt)+", "+str(offset)+"($"+str(base)+")")
    return
def sh(rt,offset,base):
    print("sb $"+str(rt)+", "+str(offset)+"($"+str(base)+")")
    return
def sw(rt,offset,base):
    print("sb $"+str(rt)+", "+str(offset)+"($"+str(base)+")")
    return

def random_match(r1,r2,r3):
    imm = random.randint(10,99)
    match random.randint(0,7):
        case 0:
            inst_calc_r(r1,r2,r3)
        case 1:
            inst_calc_i(r1,r2,imm)
        case 2:
            inst_md(r1,r2)
        case 3:
            inst_mft(r1)
        case 4:
            if(random.randint(0,1)):
                lw(r1,0,0)
            else:
                sw(r1,0,0)
        case 5:
            match random(0,2):
                case 0:
                    lw(r1,0,0)
                case 1:
                    lh(r1,random.randint(0,1)*2,0)
                case 2:
                    lb(r1,random.randint(0,3),0)
        case 6:
            match random(0,2):
                case 0:
                    sw(r1,0,0)
                case 1:
                    sh(r1,random.randint(0,1)*2,0)
                case 2:
                    sb(r1,random.randint(0,3),0)
        case 7:
            lui(r1,imm)
    return
# 2.continuous instructions
def generate():
    a = random.randint(0,31)
    b = random.randint(0,31)
    c = random.randint(0,31)

    random_match(a,b,c)
    match(random.randint(0,3)):
        case 0: random_match(b,a,c)
        case 1: random_match(b,c,a)
        case 2: random_match(c,a,b)
        case 3: random_match(c,b,a)
    match(random.randint(0,5)):
```

```
        case 0: random_match(b,a,c)
        case 1: random_match(b,c,a)
        case 2: random_match(c,a,b)
        case 3: random_match(c,b,a)
        case 4: random_match(a,b,c)
        case 5: random_match(a,c,b)
    match(random.randint(0,5)):
        case 0: random_match(b,a,c)
        case 1: random_match(b,c,a)
        case 2: random_match(c,a,b)
        case 3: random_match(c,b,a)
        case 4: random_match(a,b,c)
        case 5: random_match(a,c,b)

# 3.main logic
for i in range(0,31):
    print("ori $"+str(i)+",$0,",random.randint(10,99))
for i in range(0,100):
    generate()
```

自动化

使用魔改版Mars和cpu进行对拍。使用file-diff进行自动可视化对拍。同时设置无人值守测试。

4.思考题

1. 为什么需要有单独的乘除法部件而不是整合进 ALU？为何需要有独立的 HI、LO 寄存器？

因为乘除法具有很长的延迟，且相对独立，对其他部件的影响较小。将乘除法部件单独分离出来，可以使乘除法和其他运算并行，提高流水效率。且乘除法是时序逻辑，ALU为组合逻辑，功能相似度低，因遵循“高内聚，低耦合”的原则将二者分离。

因为HI和LO的操作自成一派，与GRF寄存器的使用大相径庭。独立出来可以使功能更加内聚。

2. 真实的流水线 CPU 是如何使用实现乘除法的？请查阅相关资料进行简单说明。

乘法：通常有若干个较小的乘法单元组成（组合逻辑），每个周期计算特定的几位，依次累加。

除法：通常使用试商法，使用组合逻辑在一个周期内计算 4 位左右的商。

3. 请结合自己的实现分析，你是如何处理 Busy 信号带来的周期阻塞的？

我在Busy或Start为1时，若D级指令为读写HILO的指令，则阻塞。

```
Stall = Stall_rs || Stall_rt || Stall_md;
```

因为读写HILO的指令必须等到乘除法结束（即Busy为0）时进行才能取到正确的值。而乘除法指令会覆盖旧的HILO值，故无需阻塞。

4. 请问采用字节使能信号的方式处理写指令有什么好处？（提示：从清晰性、统一性等角度考虑）

字节使能信号耦合了写使能和字节选择两种信号的功能，统一性高。用位的电平来表示有效位的范围，清晰简明，

5. 请思考，我们在按字节读和按字节写时，实际从 DM 获得的数据和向 DM 写入的数据是否是一字节？在什么情况下我们按字节读和按字节写的效率会高于按字读和按字写呢？

不为一字节，仍为按字读取。
如果有这样一种指令，他会读第一个字的最后一个字节的高四位，读第二个字的第一个字节低四位，拼成一个半字，那么此时效率会高。

6. 为了对抗复杂性你采取了哪些抽象和规范手段？这些手段在译码和处理数据冲突的时候有什么样的特点与帮助？

对指令进行分类，总结共性。在译码和处理数据冲突时达到了简化代码的效果。

7. 在本实验中你遇到了哪些不同指令类型组合产生的冲突？你又是如何解决的？相应的测试样例是什么样的？

P6主要增加了乘除法指令的冲突。只需要列出AT矩阵和转发矩阵，修改Stall和Forward以及转发数据的表达式即可。

测试样例如下：

```
ori $1, $0, 3
ori $2, $0, 14
mult $1, $2
mflo $3
ori $4, $0, 100
mult $3, $4
mflo $5
div $5, $2
mfhi $1
```

8. 如果你是手动构造的样例，请说明构造策略，说明你的测试程序如何保证**覆盖**了所有需要测试的情况；如果你是**完全随机**生成的测试样例，请思考完全随机的测试程序有何不足之处；如果你在生成测试样例时采用了**特殊的策略**，比如构造连续数据冒险序列，请你描述一下你使用的策略如何**结合了随机性**达到强测的效果。

构造以四为单位的连续冒险阵列。通过对操作寄存器的控制保证它们之间一定存在强相关的阻塞或转发关系。

考虑到随机测试样例在跳转方面具有很强的约束性，我在生成基础指令的强数据的基础上，手动插入跳转代码。例如，在ori所有寄存器之后，插入如下代码：

```
jal baseTestCode
nop
#跳转区
jalTest1:sb $0, 1($0)
lb $0, 1($0)
sb $0, 0($0)
sb $23, 0($0)
ori $31, $0, 0x3314
jr $31
......
beqTest1:mtlo $1
lb $23, 2($0)
sb $1, 0($0)
lui $0 45
ori $31, $0, 0x3114
jr $31
......
#基础指令区
baseTestCode:
......
```